CAPÍTULO 8

LOCALIZACIÓN VISUAL DE ROBOTS EN LA ROBOCUP MEDIANTE ALGORITMOS EVOLUTIVOS

E. PERDICES[1], J.M. CAÑAS[1], J. VEGA[1], C.E. AGÜERO[1], F. MARTÍN[1]

[1]Grupo de Robótica. Grupo de Sistemas y Comunicaciones – GSYC. Universidad Rey Juan Carlos (Madrid, España). E-mail: {eperdices,jmplaza,caguero,fmartin}@gsyc.es, julio.vega@urjc.es

Resumen. La autolocalización de robots en entornos conocidos es actualmente uno de los retos más importantes dentro del campo de la robótica. A partir de los distintos sensores con los que cuenta el robot, como sensores de ultrasonido, sensores láser o cámaras, el robot tiene que estimar su posición en el mundo que le rodea. En este contexto hemos desarrollado una nueva técnica para autolocalizar a un robot humanoide Nao dentro del campo de fútbol de la plataforma estándar de la RoboCup, utilizando sólo una cámara como sensor externo. Se ha realizado un análisis 2D de cada una de las imágenes recibidas desde la cámara utilizando visión artificial con el fin de detectar las líneas y porterías del campo. Mediante estas imágenes y el mapa del mundo que rodea al robot hemos sido capaces de extraer información de posición que nos ha permitido conocer qué posiciones son plausibles para el robot dada una observación. Se ha diseñado un algoritmo de autolocalización para acumular la información de las imágenes mediante un algoritmo evolutivo que ha sido validado experimentalmente en competiciones reales.

1 Introducción

La RoboCup es un proyecto a nivel internacional para promover la inteligencia artificial, la robótica y otros campos relacionados. Se trata de promocionar la investigación sobre robots e IA proporcionando un problema estándar donde un amplio abanico de tecnologías pueden ser integradas y examinadas. La meta final de la RoboCup es: "Para el año 2050, desarrollar un equipo de fútbol de robots humanoides totalmente autónomos capaces de jugar y ganar contra el mejor equipo humano del mundo que exista en ese momento."

Para llegar a esta meta se deben incorporar diversas tecnologías, a las que debe hacer frente cada robot, tales como razonamiento en tiempo real, utilización de estrategias, trabajo colaborativo entre robots, uso de múltiples sensores o, el campo en el que está centrado este artículo, la autolocalización visual.

Existen diversas competiciones que se centran en distintos tipos de robot que juegan al fútbol, entre las que destaca la liga de la plataforma estándar (SPL). En esta liga todos los participantes utilizan el mismo robot y sólo se centran en el desarrollo del software de éste, utilizando actualmente el robot Nao de Aldebaran Robotics (*Figura 1*). Los robots deben ser totalmente autónomos y deben utilizar una cámara como sensor principal.

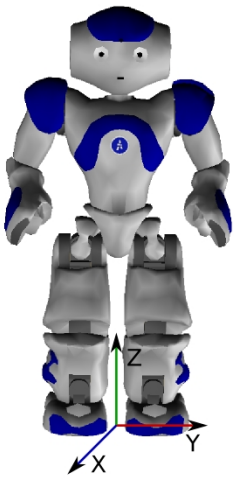

Figura 1. Robot Nao de Aldebaran Robotics.

El robot Nao es el robot utilizado en la liga de la plataforma estándar de la RoboCup desde el año 2008, donde sustituyó al robot Aibo de Sony. Se

trata de un robot humanoide desarrollado por Aldebaran Robotics que cuenta con 25 grados de libertad, lo que permiten al robot realizar un gran número de movimientos, y cuenta con múltiples sensores, tales como 2 cámaras, sensores ultrasonido, inerciales y sensores de presión (FSR), que permiten al robot obtener información del mundo que le rodea.

El robot Nao cuenta con una arquitectura software llamada Naoqi que se encarga de controlar el estado del robot. Esta arquitectura hace que el acceso a los sensores del robot sea más sencillo y proporciona distintas funciones para mover sus actuadores o para obtener medidas de los distintos sensores. La versión de Naoqi utilizada en este artículo ha sido Naoqi 1.6.0, liberada en enero de 2010.

Uno de los factores determinantes para el robot jugador es saber qué hacer cuando percibimos la pelota, puesto que la actuación del robot deberá ser distinta cuando se encuentre en su propia área (intentará despejar) que cuando esté en el área contraria (intentará tirar a portería). Por ello, uno de los problemas más importantes que se plantean es la localización dentro del campo.

Al utilizar las cámaras como sensores, podemos obtener toda la información del mundo que rodea al robot mediante visión artificial. El terreno de juega en el que se encuentran los robot nos proporciona la información necesaria para la localización del robot, ya que cuenta con dos porterías con colores distintos (azul y amarilla) y diversas líneas de color blanco: las líneas que delimitan el campo, las líneas del área, la línea de medio campo y el círculo central.

En el contexto de la RoboCup, las simetrías en las observaciones son muy comunes, por ejemplo cuando el robot sólo puede ver un poste de la portería o cuando se detectan líneas independientes. Debido a ello, los algoritmos de localización deben ser capaces de manejar estas simetrías lo mejor posible para alcanzar la solución correcta.

2 Autolocalización en la RoboCup

Para resolver el problema de la autolocalización en la SPL de la RoboCup, existen diferentes técnicas utilizadas por los equipos participantes. La técnica más utilizada son los filtros de partículas, como la localización por Monte Carlo (MCL) (*Fox, 1999*), que ha sido utilizada por distintos equipos con resultados exitosos (*Hester, 2008, Laude, 2009*).

Otra técnica bastante utilizada son los filtros de Kalman (*Kalman, 1960*), aunque esta técnica no tiene tan buenos resultados como los filtros

de partículas debido a que no es capaz de recuperarse ante secuestros. Por ello, algunos equipos mezclan esta técnica con otras como los algoritmos Markovianos (*Simmons, 1995*), como (*Martín, 2008, Brindza 2009*).

En algunos de los algoritmos basados en muestras, como es el caso de Monte Carlo, el comportamiento del algoritmo puede no ser del todo correcto cuando se producen simetrías. Esto sucede porque MCL sólo es capaz de guardar una única localización para el robot en cada iteración, por lo que si el robot se encuentra ante una observación con simetrías, el algoritmo puede seleccionar una posición incorrecta donde las observaciones sean similares. Monte Carlo solucionaría esta situación tras varias iteraciones en las que las observaciones descartasen la posición actual y se produciría un *remuestreo*.

A la hora de analizar las observaciones recibidas de la cámara del robot también se utilizan distintas técnicas por parte de los distintos equipos, aunque todas ellas tienen en común que intentan ser tan rápidas como sea posible, ya que la eficiencia es uno de los aspectos más importantes a la hora de competir en la RoboCup. Existen dos enfoques principales a la hora de detectar los elementos en la imagen, el primero trata de encontrar puntos independientes que formen parte de los elementos del campo (*Röfer, 2004*), mientras que el segundo intenta detectar elementos más complejos, como líneas, porterías, etc (*Stronger, 2007*).

En este artículo vamos a utilizar un método de análisis de la imagen basado en puntos independientes, como explicaremos en la *Sección 3.1*, dejando métodos de análisis más complejos para trabajos futuros.

3 Algoritmo evolutivo de autolocalización visual

Teniendo en cuenta las características necesarias en el contexto de la RoboCup, hemos diseñado e implementado un nuevo algoritmo especialmente diseñado para manejar simetrías en el campo de la RoboCup. Para ello, hemos desarrollado un algoritmo evolutivo, un tipo de algoritmo de optimización metaheurístico que se inspira en la evolución biológica para buscar la solución a un problema.

En este tipo de algoritmos evolutivos, las soluciones candidatas son conocidas como individuos, los cuales forman parte de una población que evoluciona a lo largo del tiempo utilizando para ello operadores genéticos, como por ejemplo la mutación o el cruce, que explicaremos más adelante. Además, cada individuo es evaluado a partir de una función de calidad que

nos permite conocer su "salud", es decir, saber en qué medida se acerca a la solución real.

En nuestra implementación, el algoritmo evolutivo cuenta con 4 elementos básicos:

- Individuos: Los individuos guardan una posición del robot (X, Y, θ) (*Figura 2*) y la probabilidad obtenida de la función salud en la última iteración.
- Razas: Cada raza es una población de individuos que representan una posible solución a la localización del robot. En el algoritmo existen varias razas que compiten entre sí para ser la ganadora en una iteración determinada, donde la raza ganadora es considerada la posición actual del robot.
- Exploradores: Individuos independientes encargados de buscar posiciones en las que generar nuevas razas.
- Explotadores: Cada uno de los individuos que forman parte de una raza, encargados de analizar en profundidad una posible posición en busca de la mejor solución.

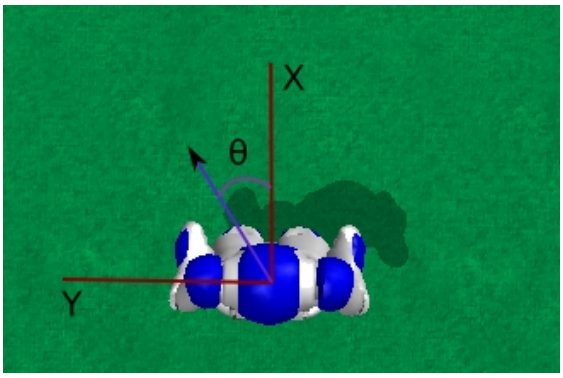

Figura 2. Posición y orientación del robot.

El comportamiento general del algoritmo consiste en mantener varias razas que compiten entre sí en distintas posiciones del campo. En el caso de que se produzcan simetrías surgirán razas en cada una de las posiciones en las que se pueda encontrar el robot, hasta que, previsiblemente, tras varias iteraciones se obtenga información de las observaciones que descarte a las posiciones incorrectas y se seleccione la posición real del robot.

En cada iteración del algoritmo se realizan varios pasos que vamos a explicar en las siguientes secciones.

3.1 Función salud

A pesar de que el robot Nao cuenta con dos cámaras, hemos decidido utilizar solamente una de ellas (la cámara inferior) debido a que no es posible realizar visión estéreo con ellas, y que el hecho de cambiar de una cámara a otra dependiendo de lo que queramos ver es muy costoso computacionalmente y aumenta la complejidad de los algoritmos.

Nuestro modelo de observación analiza la imagen de entrada y obtiene puntos característicos en la imagen en 2D. Para conseguir estos puntos, hemos realizado un barrido asimétrico de la imagen en horizontal y vertical (*Figura 3*). Así, la zona superior de la imagen se analiza más en profundidad que la parte inferior, puesto que los objetos más alejados se encontrarán en la parte superior de la imagen y tendrán una menor resolución.

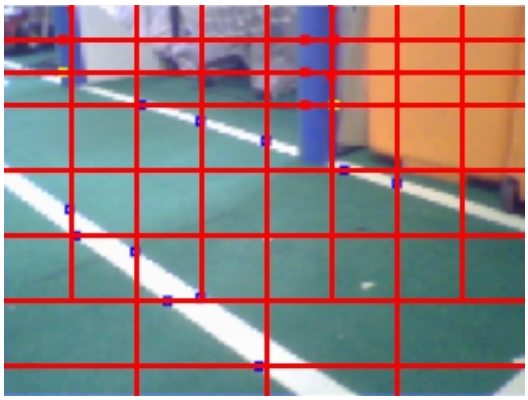

Figura 3. Barrido en rejilla de la imagen.

A lo largo de cada una de las rectas verticales y horizontales con las que recorremos la imagen, realizamos para cada píxel varios filtros de color con el fin de identificar el color del píxel que estamos analizando. Los filtros de color realizados se corresponden con los distintos elementos que podemos encontrarnos en el campo, así, se identifican los colores amarillo y azul para las porterías, el color blanco para las líneas, el naranja para la pelota, y el verde para el campo.

Al realizar esta identificación en cada recta, buscamos los puntos donde se produce un cambio de color, detectando de esta forma los bordes en la imagen. Así pues, para encontrar por ejemplo la portería azul, nos quedaremos con los píxeles donde haya un cambio de color entre el azul y cualquier otro color. Todos los píxeles que cumplan esta condición serán almacenados como puntos característicos en la imagen.

Entre los puntos obtenidos pueden encontrarse falsos positivos que harían que el algoritmo se comportase de un modo incorrecto, por ello, el siguiente paso a realizar es intentar encontrar estos falsos positivos y eliminarlos, consiguiendo así que los puntos seleccionados sean más fiables. Para ello, los puntos deben cumplir las siguientes características:

1. Un punto no puede estar muy cerca de otro del mismo color.
2. No puede existir un punto aislado, es decir, debe estar junto a más píxeles del mismo color.
3. Los puntos pertenecientes a líneas deben estar por debajo del fin del campo (donde finaliza el color verde).
4. Los puntos de las porterías deben estar dentro de una región con un determinado tamaño y forma.
5. Un punto no puede formar parte del propio cuerpo del robot.
6. Se debe detectar si el punto pertenece a un obstáculo (por ejemplo, otro robot).

Tras todos estos filtros, es posible que aún tengamos un gran número de puntos que harían que aumentase el tiempo de cómputo del algoritmo. Por ello, hemos establecido un número máximo de puntos para las líneas (12) y para las porterías (8), que son seleccionados de forma aleatoria entre los puntos válidos hasta ese momento (*Figura 4*).

Figura 4. Puntos característicos seleccionados, en azul puntos de líneas, en rojo puntos de porterías y en amarillo puntos aleatoriamente descartados.

Con los puntos finalmente seleccionados, tenemos que ser capaces de calcular la salud de cada individuo. Con este propósito, necesitamos comparar la observación actual del robot (imagen real) con la imagen que obtendríamos si el robot se encontrase en una cierta posición (imagen teórica). Así, si ambas imágenes son parecidas esa posición es plausible, mientras que si son muy distintas esa posición es improbable.

Las imágenes se comparan utilizando los puntos seleccionados en la imagen real. Para cada punto seleccionado en la imagen, buscamos en la imagen teórica el punto más cercano del mismo color, lo que nos proporciona una distancia *d* en píxeles que nos servirá para calcular la probabilidad final (*Figura 5*). Al realizar el mismo procedimiento para cada punto detectado en la imagen real, podremos calcular la probabilidad final mediante la media de estas distancias, como se ve en la siguiente ecuación:

$$H = 1 - \frac{\sum_{i=0}^{N} \frac{d_i}{N}}{M} \qquad (1)$$

donde *N* es el número de puntos y *M* es una constante que se ha establecido en 50 para una imagen de tamaño 160x120.

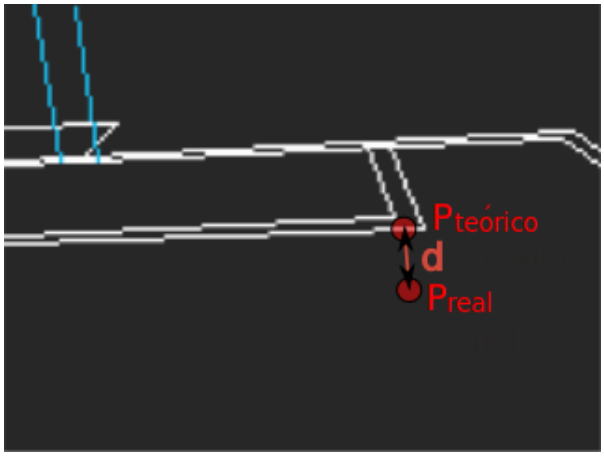

Figura 5. Distancia entre puntos en la imagen real y la teórica.

Para calcular la distancia *d* entre la imagen real y la teórica podríamos generar la imagen teórica ad-hoc para cada individuo, sin embargo, esto requeriría mucho tiempo de cómputo, especialmente en un robot como el

Nao que cuenta con unos recursos muy limitados y donde los algoritmos deben ser lo más eficientes posible.

Por ello, hemos precalculado cierta información en memoria que nos permitiese mejorar la eficiencia. Así, almacenamos la correspondencia en 3D entre cada posición del campo (cada 2,5 cm) y su línea más cercada y cada posición en el plano de la portería y su poste más cercano.

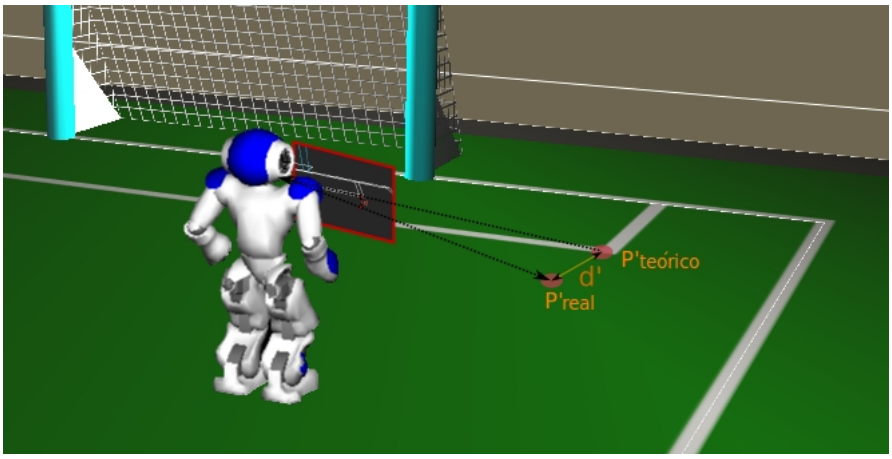

Figura 6. Calculando el punto $P_{teórico}$ con información precalculada.

Así, podemos calcular el punto $P_{teórico}$ en la *Figura 5* como vemos en la *Figura 6*. Dado el mismo punto P_{real} seleccionado, retroproyectamos este punto 2D en el plano seleccionado para obtener el punto en 3D P'_{real}. Después, consultamos el punto precalculado en 3D más cercano a P'_{real}, y así obtenemos $P'_{teórico}$. Este $P'_{teórico}$ en 3D lo proyectamos en la imagen y obtenemos el punto $P_{teórico}$ en 2D que buscábamos, pudiendo así calcular la distancia entre ellos.

3.2 Creación de exploradores

El primer paso del algoritmo consiste en evaluar nuevos exploradores para encontrar nuevas posiciones en las que el robot tendría una salud muy alta. Estos exploradores pueden ser repartidos por el campo de forma aleatoria, sin embargo , ya que el campo de la RoboCup es muy pequeño, el algoritmo muestra un mejor comportamiento si se inicializan los exploradores en posiciones predeterminadas.

Estos exploradores son ordenados según su salud de mayor a menor. Como este procedimiento suele tener un alto coste computacional (se evalúan 490 candidatos), hemos dividido este cálculo en dos iteraciones.

Tras ello, seleccionamos los mejores 6 exploradores y realizamos una pequeña búsqueda local para mejorar su localización. Así, partiendo de cada uno de los exploradores seleccionados, se crean 34 exploradores con distinta posición y orientación en un radio de 30 cm. Este procedimiento se realiza durante 2 iteraciones, evaluando otros 420 nuevos exploradores en total.

Tras 4 iteraciones, seleccionamos los mejores 6 candidatos evaluados y los guardamos como candidatos para llegar a ser una nueva raza, como explicaremos en la *Sección 3.3*.

La creación de exploradores conlleva un gran tiempo de cálculo, por lo que no puede ser ejecutada en todas las iteraciones. Por ello, y con el objetivo de disminuir el tiempo de ejecución del algoritmo, hemos decidido realizar la creación de exploradores únicamente cuando veamos alguna de las porterías, puesto que es el elemento del campo que más información nos proporciona para conocer la posición del robot.

3.3 Gestión de razas

Al disponer de varias razas, tenemos que desarrollar un mecanismo capaz de gestionar su creación, eliminación o fusión. Antes de detallar estos mecanismos vamos a explicar dos de los parámetros de los que dispone cada raza, el número de victorias y la vida:

- Número de victorias: En un principio, todas las razas comienzan con 0 victorias. Este número puede aumentar o disminuir en función de la salud de la raza en cada iteración, como explicaremos en la *Sección 3.6*. La raza con mayor número de victorias será la que determine la posición del robot calculada en cada iteración.
- Vida: Este parámetro determina el número de iteraciones que quedan hasta que la raza pueda ser eliminada. La vida ha sido creada con dos objetivo, dar prioridad a las razas que ya existen frente a las que se van a crear nuevas y evitar borrar razas debido a una única mala observación.

3.3.1 Creación de razas

Cuando dispongamos de nuevos candidatos a ser raza, a través de la creación de exploradores, debemos decidir si creamos nuevas razas o si sustituimos las ya existentes.

El primer paso consiste en comprobar si ya existen suficientes razas creadas, con este fin, existe un máximo número de razas *N* (actualmente fijado en 4), para evitar que el tiempo de ejecución del algoritmo sea demasiado alto. Si el número de razas es menor que *N*, significa que podemos crear nuevas razas sin necesidad de sustituir las existentes.

En el caso de que el número de razas sea *N*, debemos comprobar si las razas actuales pueden ser reemplazadas, es decir, no podemos remplazar una raza si su vida o su número de victorias es mayor que 0. Si ninguna raza puede ser reemplazada, los candidatos actuales son ignorados (en realidad, no se ejecuta la creación de exploradores).

En el caso de que podamos crear o reemplazar nuevas razas, se comprueba si la posición y orientación de los candidatos es similar a la de alguna de las razas ya existentes, para evitar crear varias razas en la misma posición.

Finalmente, si todas estas condiciones se cumplen, se crear nuevas razas hasta que alcancemos *N* razas o se reemplaza una raza en el caso de que la salud del candidato sea mayor que la salud de la raza.

3.3.2 Fusión de razas

Cuando dos razas evolucionan hacia dos localizaciones similares, tanto en posición como en ángulo, se considera que ambas razas llevan hacia la misma solución final y se produce una fusión entre ellas. Esta fusión consiste en eliminar la raza que menos victorias tenga, y en caso de empate, se eliminará la que tenga una salud más baja.

3.3.3 Evolución de razas

Cuando se crea una raza desde un explorador, todos los explotadores de la raza son creados de forma aleatoria mediante un ruido térmico que se aplica sobre el explorador que creó la raza. A partir de ese momento, en el resto de iteraciones los explotadores de la raza evolucionan utilizando tres operadores genéticos:

- Elitismo: Seleccionamos los N mejores explotadores de cada raza, que se clasificarán por su salud. Los explotadores elitistas se guardarán sin modificación para la siguiente población.
- Cruce: Se seleccionan al azar dos explotadores de la raza y se cruzan, haciendo la media entre ellos en (X, Y, θ).
- Mutación: Seleccionamos un explotador aleatoriamente y aplicamos un ruido térmico que modifica su posición y su ángulo.

Una vez evolucionados los explotadores, se calcula el valor final de la raza como la media de todos sus elitistas, haciendo que no se produzcan cambios bruscos en la localización.

Además, aplicamos a todas las razas y explotadores un operador de movimiento al inicio de cada iteración, como vamos a mostrar en la próxima sección.

3.5 Operador de movimiento

En caso de que el robot se haya movido desde la última iteración, necesitamos aplicar ese movimiento a la posición (X, Y, θ) de todas las razas y explotadores.

Para ello, necesitamos calcular el movimiento del robot entre la última iteración del algoritmo y el momento actual. Calcular este movimiento en un robot con patas no es fácil, ya que la odometría del robot cuenta con un gran error. Para hacer esta tarea más fácil, hemos utilizado una función proporcionada por Naoqi llamada *getPosition* para obtener la posición (X, Y, θ) del robot desde que éste se inició.

En cada iteración, guardamos la posición actual que devuelve la función *getPosition*. De esta forma podemos comparar la diferencia entre la posición que tenía en la última iteración (P_l) y la actual (P_n), con lo que calculamos P_d:

$$P_d = (P_n - P_l) * \epsilon \qquad (2)$$

donde ε es el error sistemático en la odometría del robot.

El siguiente paso consiste en calcular el movimiento real relativo entre dos iteraciones (P_r). Así, el giro del robot ($P_{r\theta}$) es directamente la diferencia $P_{d\theta}$ calculada, mientras que para calcular la posición (P_{rx}, P_{ry}) del ro-

bot, necesitamos aplicar una matriz RT teniendo en cuenta la rotación del robot en la última iteración ($P_{l\theta}$):

$$\begin{pmatrix} P_{r_x} \\ P_{r_y} \end{pmatrix} = \begin{pmatrix} cos(-P_{l_\theta}) & -sen(-P_{l_\theta}) \\ sen(-P_{l_\theta}) & cos(-P_{l_\theta}) \end{pmatrix} * \begin{pmatrix} P_{d_x} \\ P_{d_y} \end{pmatrix} \quad (3)$$

Una vez que ya hemos calculado este movimiento relativo, podemos actualizar la posición actual de cada individuo (P_i) para conseguir su nueva posición (P_f). La nueva rotación $P_{f\theta}$ se calcula mediante la ecuación:

$$P_{f_\theta} = P_{r_\theta} + P_{i_\theta} \quad (4)$$

mientras que para obtener los valores (P_{fx}, P_{fy}) es necesario aplicar una nueva matriz RT:

$$\begin{pmatrix} P_{f_x} \\ P_{f_y} \\ 1 \end{pmatrix} = \begin{pmatrix} cos(P_{i_\theta}) & -sen(P_{i_\theta}) & P_{i_x} \\ sen(P_{i_\theta}) & cos(P_{i_\theta}) & P_{i_y} \\ 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} P_{r_x} \\ P_{r_y} \\ 1 \end{pmatrix} \quad (5)$$

3.6 Raza ganadora

Tras evaluar todas las razas existentes debemos seleccionar una de ellas para que sea la localización final del robot. La raza seleccionada será la que cuente con mayor número de victorias tras completar la iteración, por lo que en cada iteración debemos aumentar o disminuir este valor en cada una de las razas.

El primer paso a realizar es seleccionar la raza con mayor salud en esta iteración (R_i). En el caso de que la raza seleccionada ya fuese la raza ganadora en la anterior iteración (R_a), se aumentará el número de victorias de R_i y se disminuirá el del resto de razas. Sin embargo, si R_i y R_a son diferentes, sólo cambiaremos los valores del número de victorias cuando la diferencia entre la salud de R_i y la de R_a sea suficientemente grande.

Esto es así puesto que nos interesa que sea difícil cambiar la raza ganadora cuando la imagen observada no sea muy buena (no se vea ninguna

portería), o si R_a ha sido la raza ganadora durante muchas iteraciones. De esta forma, intentamos favorecer a las razas que ganen durante muchas iteraciones y sólo cambiamos cuando la diferencia de salud sea muy grande (lo que indicaría que la localización actual no es correcta).

Al final de la iteración, la nueva raza ganadora sera la raza con mayor número de victorias, y su posición determinara la posición del robot calculada por el algoritmo.

4 Experimentos

Hemos realizado numerosos experimentos para validar la función de salud y el algoritmo evolutivo tanto en simuladores como en el robot real.

4.1 Discriminación de función salud

Para validar la función salud hemos implementado un mecanismo de depuración que permite ver el valor devuelto por esta función en distintas posiciones. En la *Figura 9* mostramos el valor obtenido en todas las posiciones (X, Y) del campo, donde las zonas de color rojo son las que más probabilidad tienen y las blancas las que menos.

Además, en la *Figura 10* mostramos el valor obtenido para cada orientación θ tras seleccionar la posición correcta del robot.

Como podemos ver, en ambas imágenes la posición y la rotación con más probabilidad son plausibles con la imagen de entrada. El análisis de la imagen tarda 1.5 ms en el robot real, y la función salud se calcula en 100 microsegundos para cada posición.

El algoritmo ha sido probado en otras situaciones, como frente a simetrías, oclusiones o falsos positivos. En el caso de las simetrías y las oclusiones el algoritmo tiene un comportamiento favorable, ya que nuestro algoritmo no penaliza si algunos de los objetos no son detectados en la imagen, y el único punto negativo es que aumenta la probabilidad en ciertas zonas del campo.

Sin embargo, los faltos positivos afectan negativamente a la salud de los individuos y la localización calculada puede ser totalmente errónea, por esta razón hemos intentado evitar los falsos positivos en el análisis de la imagen con numerosos filtros, como ya explicamos en la *Sección 3.1*.

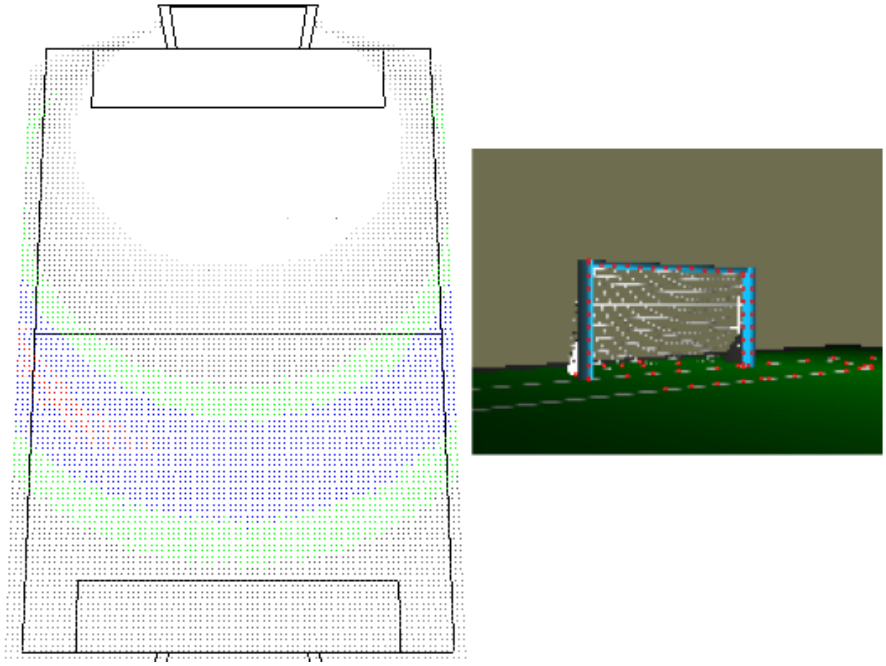

Figura 9. Función salud en distintas posiciones (X, Y).

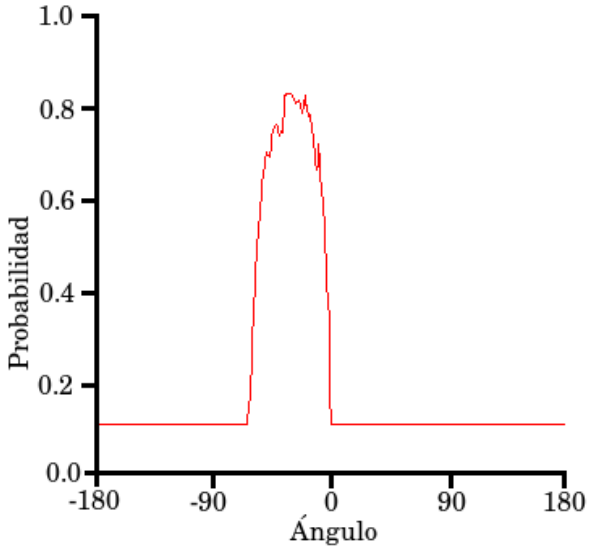

Figura 10. Función salud con distintas orientaciones.

4.2 Convergencia del algoritmo

Hemos realizado de nuevo varios experimentos para validar el algoritmo evolutivo encargado de acumular observaciones, especialmente en el robot real. El algoritmo cuenta con varios parámetros que deben ser configurados, como el número máximo de razas, los explotadores en cada raza, el porcentaje de elitistas en cada raza y el porcentaje de individuos que evolucionan mediante cruce.

El número máximo de razas ha sido un factor fundamental, puesto que a mayor número de razas, mejores serían los resultados en cuanto a precisión de posición pero aumentaría el tiempo de ejecución de forma lineal. Debido a esto, decidimos quedarnos en un valor medio en el que la eficiencia y la precisión estuviesen compensadas, con 4 razas.

Así, el coste computacional medio del algoritmo en el robot real ha sido de 37.6 milisegundos por iteración, aunque este tiempo cambia en función del número de razas y de si lanzamos exploradores en esa iteración o no, pudiendo ir desde 10 ms hasta 80 ms.

El primer experimento (*Figura 11*) consiste en situar al robot en el centro del campo y colocar distintos puntos de referencia a los que el robot debe ir. En estos puntos de referencia medimos el error que se produce entre la posición calculada y la posición real del robot.

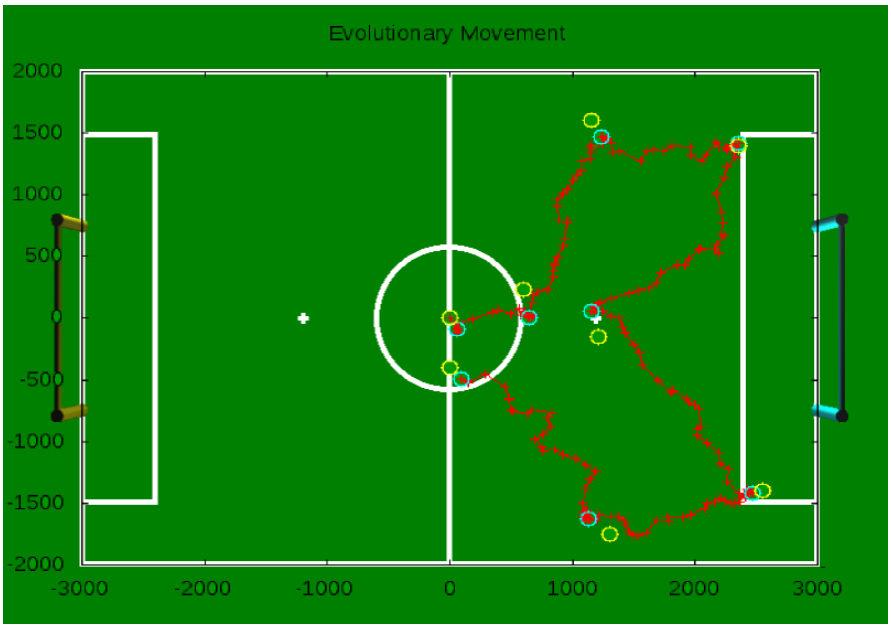

Figura 11. Seguimiento del movimiento del robot en el campo.

En ella mostramos la trayectoria calculada por el algoritmo (rojo), la localización calculada en los puntos de referencia (círculos azules) y la posición real del robot (círculos amarillos).

El algoritmo es capaz de seguir el movimiento que realiza el robot con un error medio de 14.3 cm y 8.42 grados. Además, podemos destacar que la trayectoria seguida por el robot es muy estable y está siempre cercana a la que sigue el robot en la realidad.

4.3 Recuperación ante secuestros

El segundo experimento (*Figura 12*) consiste en secuestrar al robot en numerosas ocasiones y comprobar el error medio en cada secuestro. Esto es muy común en la RoboCup puesto que los árbitros pueden sancionar a los robots y moverles de posición en función de la situación.

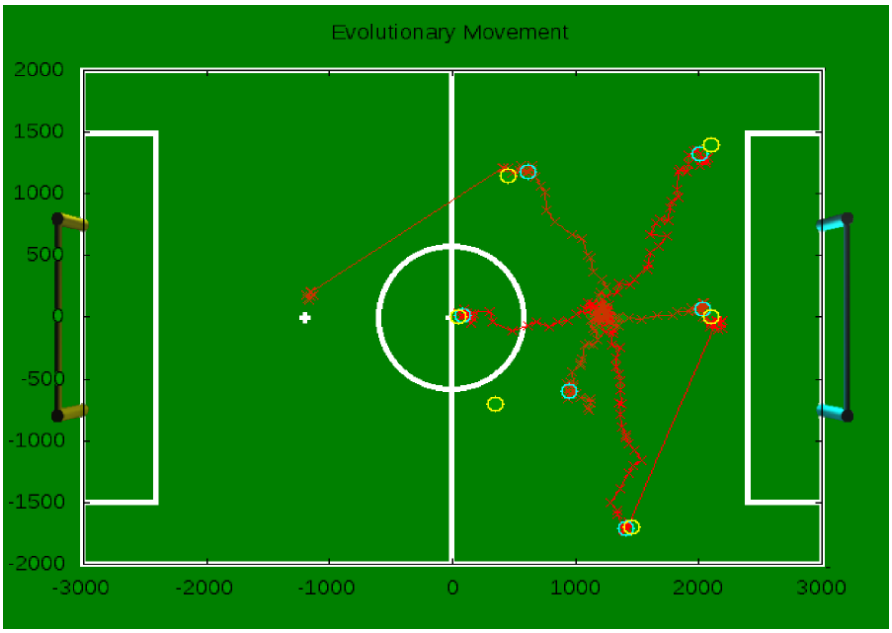

Figura 12. Secuestros con algoritmo evolutivo.

En este experimento, hay dos situaciones que debemos destacar:

- La primera es el cambio de razas que se produce al comienzo de algunos secuestros, como se puede ver en dos de ellos. Normalmente, si se producen simetrías en las observaciones, se crearán varias razas

con una salud similar, lo que puede hacer que la raza ganadora cambie entre ellas. Tras ello, el algoritmo normalmente recibirá nueva información que descartará las posiciones incorrectas y hará que el cambio entre razas finalice.

- La segunda situación a destacar es el error que se produce en uno de los 6 secuestros realizados, que es mucho mayor al de los demás. Esto sucede debido nuevamente a las simetrías, sin embargo, el error no es demasiado grande debido a que el algoritmo está preparado para ello, con lo que podrá recuperarse cuando reciba información más concluyente.

En este caso, el error medio calculado en las posiciones iniciales es de 17.5 cm y 6.36 grados, teniendo en cuenta el caso de mayor error debido a las simetrías.

También hemos medido el tiempo medio que tarda el algoritmo en calcular una nueva posición plausible después de un secuestro (tiempo de recuperación), que ha sido de 20.6 segundos en este experimento.

El tiempo de recuperación no es muy grande ya que también utilizamos los sensores de presión FSR del robot para notar cuando el robot no está tocando el cuelo, lo que provoca que el algoritmo se reinicie. En caso de que no utilizásemos estos sensores, el tiempo de recuperación medio sería de 30 segundos, es decir, un 50% mayor.

5 Conclusiones

En este artículo hemos presentado una nueva alternativa para localizar robots en entornos conocidos cuando en la localización pueden encontrarse simetrías en el mundo.

Hemos explicado y probado este nuevo algoritmo en entornos reales y simuladores para validar experimentalmente su comportamiento, con unos resultados satisfactorios.

El error medio medido en el robot real ha sido menor a 20 centímetros, lo que demuestra que nos encontramos ante un algoritmo robusto que puede ser utilizado en condiciones reales. Además, al estar especialmente diseñado para poder manejar simetrías, las probabilidades de que el robot se encuentra en una posición incorrecta debido a éstas es muy bajo, al contrario de lo que sucede con otros algoritmos clásicos utilizados en la RoboCup.

Como punto negativo, este algoritmo es más complejo y requiere un tiempo de ejecución mayor que otros algoritmos, aunque siempre dentro de niveles razonables.

El trabajo realizado puede ser continuado por varias vías. La primera de ellas es la mejora en el análisis de la imagen, detectando objetos en lugar de puntos, como líneas, porterías, intersecciones entre líneas, etc, lo que haría que el número de simetrías se redujese y la localización calculada fuese más fiable. Además, podemos probar el algoritmo en entornos menos controlados que el campo de la RoboCup, como por ejemplo un edificio de oficinas.

Referencias

Fox, D., Burgardy, W., Dellaert, F. y Thrun, S., Monte Carlo Localization: Efficient Position Estimation for Mobile Robots. In Proc. of the National Conference on Artificial Intelligence, 1999.

Hester, T. y Stone, P., Negative Information and Line Observations for Monte Carlo Localization. The IEEE International Conference on Robotics and Automation, 2008.

Laue, T., Jeffry de Haas, T., Burchardt, A., Graf, C., Röfer, T., Hartl, A. y Rieskamp, A., Efficient and Reliable Sensor Models for Humanoid Soccer Robot Self-Localization. The 2009 IEEE-RAS Intl. Conf. On Humanoid Robots (Humanoids 2009), 2009.

Kalman, R. Emil., A new approach to linear filtering and prediction problems. Transactions of the ASME-Journal of Basic Engineering, 1960.

Simmons, R. y Koening, S., Probabilistic navigation in partially observable environments. In Proceedings of the 1995 International Joint Conference on Artificial Intelligence, 1995.

Martin, F., Aportaciones a la auto-localizacion visual de robots autonomos con patas. PhD Thesis, Universidad Rey Juan Carlos, 2008.

Brindza, J, Lee, A., Majumdar, A., Scharfman, B., Schneider, A., Shor, R. y Lee, D., UPennalizers RoboCup Standard Platform League Team Report. Technical report, 2009.

Röfer, T. y Jungel, M., Fast and Robust Edge-Based Localization in the Sony Four-Legged Robot League. In 7th International Workshop on RoboCup 2003, 2004.

Stronger, D. y Stone, P., A Comparison of Two Approaches for Vision and Self-Localization on a Mobile Robot. In International Conference on Robotics and Automation, 2007.

Vega, J., Cañas, J.M., Sistema de atención visual para la interacción persona-robot. In Workshop on interacción persona-robot, RoboCity 2030, pp. 91-110, 2009